Chapter 10

File Systems

Given that main memory is volatile, i.e., does not retain information when power is turned off, and is also limited in size, any computer system must be equipped with secondary memory on which the user and the system may keep information for indefinite periods of time. By far the most popular secondary memory devices are disks for random access purposes and magnetic tapes for sequential, archival storage. Since these devices are very complex to interact with, and, in multi-user systems are shared among different users, operating systems provide extensive services for managing data on secondary memory. This data is organized into *files*, which are collections of data elements grouped together for the purposes of access control, retrieval, and modification.

A *file system* is the part of the operating system that is responsible for managing files and the resources on which these reside. Without a file system, efficient computing would essentially be impossible. This chapter discusses the organization of file systems and the tasks performed by its different components. The first part is concerned with general user and implementation aspects of file management emphasizing centralized systems; the last sections consider extensions and methods for distributed systems.

10.1 Basic Functions of File Management

The file system, in collaboration with the I/O system, has the following three basic functions:

1. Present a *logical* or *abstract view* of files and directories to the users by

hiding the physical details of secondary storage devices and the I/O operations for communicating with the devices.

2. Facilitate *efficient use* of the underlying storage devices.

3. Support the *sharing of files* among different users and applications. This includes providing *protection* mechanisms to ensure that information is exchanged in a controlled and secure manner.

The first function exists because physical device interfaces are very complex. They also change frequently as new devices replace outdated ones. The I/O system, discussed in Chapter 11, provides the first level of abstraction on top of the hardware devices. It presents an interface where devices may be viewed as *collections or streams of logical blocks*, which can be accessed sequentially or directly, depending on the type of device.

This level of abstraction is still too low for most applications, which need to manipulate data in terms of named collections of data records—**files**—and organize these into various hierarchical structures using **directories** (sometimes called **folders**). Thus the role of the files system is to extend the logical I/O device abstraction to a file-level abstraction.

A file is defined as a set of related data items, called **logical records**. It is largely independent of the medium on which it is stored. The logical record is the smallest addressable unit within a file. The purpose of the file concept is to give users one simple uniform linear space for their code and data. Files and records within files may then be manipulated via a set of high-level operations defined by the **file system interface**, and whose implementation is hidden from the user by the file system.

The second goal and function of a file system is the efficient use of the storage devices on which files that belong to potentially many different users are kept. Unlike main memory, which has no moving parts and where access to any location takes the same amount of time, most secondary memory devices are electro-mechanical, are much slower than main memory, and have data access times that depend greatly on the data location on the device. For example, a disk access may or may not require a seek operation, i.e., the physical movement of the read/write head, depending on the head's current position. Since each seek is very time consuming, the file system must strive for placing data items in such a way as to minimize the read/write head movements. As discussed in Chapter 11, the I/O system may then provide additional optimizations, for example by dynamically reordering the disk accesses to minimize the distance traveled by the read/write head. The

Figure 10.1: A Hierarchical view of the File System

situation is even more critical with tapes and other one-dimensional storage devices, where data can be accessed effectively only in the order in which they appear on the tape. The file system must assure that a data placement appropriate for the underlying storage device is chosen for any given file.

The third reason for providing a file system is to facilitate file sharing. This is similar to the sharing of code and data in main memory, where two or more processes have simultaneous access to the same portion of main memory. The main difference is that files, unlike main memory information, generally persist in the secondary memory even when its owner or creator is not currently active; for example, the owner may be logged out. Thus sharing of files does not have to be concurrent—a process may wish to access a file written earlier by some other process that no longer exists. The key issue is protection, that is, controlling the type of access to a file permitted by a given process or user.

In this chapter we are concerned primarily with the first two functions of a file system, which address the need for making details of the physical storage devices transparent to users and their programs. The problems of protection and security are treated separately in Chapters 12 and 13.

10.2 Hierarchical Model of a File System

The file system, like any other complex software system, can be decomposed into several distinct parts according to their primary functions. We will adopt a hierarchical organization, where each level represents a successively more abstract machine in that a module at a given level may call upon the service of only modules at the same or lower levels. While not all file systems are designed according to a strict hierarchy, this conceptual view is useful because it permits us to understand the complex functionalities of the file system and its interfaces to both the user and the I/O system by studying each level in isolation.

Figure 10.1 shows the organization. The left-hand side illustrates the file system's position as a layer between the user and the I/O system. For the user, it creates the abstraction of logical files, which can be manipulated and organized using the various file and directory commands presented in Sections 10.3.4 and 10.4.2. It manages these on the underlying disks and other

secondary memory devices by calling upon the services of the I/O system and exchanging data with it using logical block numbers. The responsibility of the I/O system is to translate the logical block numbers into actual disk or tape addresses.

The remainder of Figure 10.1 shows the file system subdivided into three main components in a hierarchy. Below, we briefly outline the tasks within each level, proceeding from the abstract user interface down to the file system and I/O interface. Succeeding sections then describe the functions of each level in more detail.

The File System Interface. The file system presents the abstraction of a **logical file** to the user, and defines a set of operations to use and manipulate these. It also provides a way to organize individual files into groups and to keep track of them using **directories**. Typically, a file can be identified using two forms of names, depending on the operation to be performed on that file. A **symbolic name**, chosen by the user at file creation, is used to find a file in a directory. It is also used when manipulating the file as a whole, for example, to rename or delete it. Symbolic names, however, are not convenient to use within programs, especially when used repeatedly. Thus operations that access the file's data, notably, read and write operations, use a numeric identifier for the file. This is generated by lower levels of the file system. When a desired file is first opened by setting up data structures necessary to facilitate its access, an identifier, the **open file id** (Figure 10.1), is returned and used for subsequent read, write, and other data manipulation operations. Note that the symbolic name remains valid even when the file is not in use, while the open file id is only temporary. It is valid only while the process that opened the file remains active or until the file is explicitly closed.

Directory Management. The primary function of this level is to use the symbolic file name to retrieve the **descriptive information** necessary to manipulate the file. This information is then passed to the Basic File System module, which opens the file for access and returns the corresponding open file id to the Directory Management module. Depending on the user-level command being processed, the open file id may be returned to the user for subsequent read/write access to the file, or it may be used by the Directory Management Module itself. For example, a search through the hierarchy structure requires that the Directory Management repeatedly opens and reads subdirectories to find a desired file. At each step of the

search, the Directory Management routines must call upon the lower level Basic File System to open these subdirectories.

Basic File System. This part activates and deactivates files by invoking **opening** and **closing** routines. It is also responsible for verifying the **access rights** of the caller on each file request. The Basic File System maintains information about all open files in main memory data structures called **open file tables** (OFT). The open file id that is returned to the caller when a file is opened points directly to the corresponding entry in an open file table. Using this id instead of the symbolic file name allows subsequent read and write operations to bypass the Directory Management and Basic File System modules when accessing the file's data.

Device Organization Methods. This module performs the mapping of the logical file to the underlying blocks on the secondary memory device. It receives read and write requests from the higher level routines or the user interface. The Device Organization Methods translate each request into the corresponding block numbers and pass these on to the underlying I/O system, which then carries out the actual data transfers between the device and the caller's main memory buffers. The allocation and deallocation of storage blocks and main memory buffers is also handled at this level.

Case Study:

To illustrate the above concepts, we consider the Unix file system. Before a file may be accessed it must be opened using the following command:

```
fd = open(name, rw, ...)
```

where *name* is the file's symbolic name and *rw* is a flag specifying whether the file is to be used for reading, writing, or both. The Directory Management routines verify that the named file exists and that the user is permitted to access it using the specified read/write mode. If this is the case, the file is opened by the routines corresponding to the Basic File System, which create a new entry in the open file table. The index into this function is returned to the caller as the value of *fd*. This identifier is then used to read or write the file's contents.

The read command has the following form:

```
stat = read(fd, buf, n)
```

where *fd* is the integer identified returned by the *open* command, *buf* is the pointer to an input buffer in the caller's main memory, and n is the number of characters to be read. When the operation completes, the value of *stat* indicates the status of the read operation upon completion; it returns the number of characters actually read or a -1 if an error occurred. The *read* (and the corresponding *write*) commands are performed by the level corresponding to the Device Organization Methods of Figure 10.1.

□

10.3 The User View of Files

From the user's point of view, a file is a named collection of data elements that have been grouped together for the purposes of access control, retrieval, and modification. At the most abstract, logical level, any file is characterized by its **name**, its **type**, its logical **organization**, and several additional **attributes**.

10.3.1 File Names and Types

What constitutes a legal file name varies between different file systems. In older systems, the name length was limited to a small number of characters. For example, MS-DOS allows up to eight characters, while older versions of Unix supports 14 characters. Certain special characters are frequently disallowed as part of the files name because they are reserved to serve a special role within the operating system. This includes in particular the blank, since this is usually a delimiter for separating commands, parameters, and other data items.

More recent systems have relaxed these restrictions, allowing file names to be arbitrary strings of characters found on most keyboards, and supporting name lengths that, for most practical purposes, are unlimited. For example, Windows allows up to 255 characters, including the blank and certain punctuation characters.

Many systems, including MS-DOS or Windows 95, do not differentiate between lower and upper case characters. Thus *MYFILE, Myfile, MyFile, myfile*, or any other combination of the upper and lower case letters all refer to the same file. In contrast, Unix and Linux differentiate between lower and upper case, and thus the above name would each refer to a different file.

An important part of a file name is its **extension**, which is a short string of additional characters appended to the file name. In most systems it is separated from the name by a period. File extensions are used to indicate the file's **type**. For example, *myfile.txt* would indicate that the file is a text tile, while *myprog.bin* would indicate that this file is an executable binary program (a load module). A typical file extension is between one and four characters in length. Older systems, e.g., MS-DOS, are more rigid, requiring an extension of one to three characters, while Unix, Linux, and many other more recent systems allow more flexibility.

The file type implies the *internal format* and the *semantic meaning* of the file's contents. In the simplest case, the file is a sequence of ASCII characters with no other format imposed on it. Such files are generally called *text* files and carry the extension *.txt*. A file containing a source program generally carries the extension indicating the programming language. For example, *main.c* might be a C program while *main.f77* would be a Fortran 77 program. When compiled, the resulting files are object modules, denoted by the extension *.o* or *.obj*. A linker then takes an object module and produces the corresponding load module, a binary file, either annotated with the extension *.bin, .com*, or, as in the case of Unix, left without an extension.

The file extension generally denotes the type of the file; the type, in turn, determines the kinds of operations that may be meaningfully performed on it by a program. Note however that it is the file type and *not* the extension that determines the file's semantic meaning. In most cases, the extension is only an annotation for the *convenience* of the user and the various applications using the files. That is, most systems allow the extension to be changed at will. But that does not change the type of the file, which is inherent to its internal format. A compiler could still process a source code file, even if its extension was changed. But the compiler would fail when presented with an object module (even if this had the proper extension), since only a linker is able to correctly interpret the contents of an object module.

The file's type is usually stored in the form of a file **header**. For example, any executable Unix file must begin with a specific "magic number"—a code indicating that this is indeed an executable file. This is followed by information on how to find the code, the data, the stack, the symbol table areas, the starting point within the code, and other information needed in order to load and execute this file.

How many different file types are supported and how rigidly the correspondence between the file type and the file name extension is enforced depends on the operating system. Every system must support a minimal

set of file types, including text files, object files, and load module files. In addition, the file system must be able to distinguish between ordinary files and file directories. Other utilities and applications then establish and follow their own conventions regarding file types, and their internal formats and extensions. For example, a *.doc* file would normally be a formatted text document understandable by *MS Word*, a *.ps* would be a file understandable by a *Postscript* printer, and a *.html* would be a file containing information in the *Hypertext Markup Language* understandable by a *Web* browser.

Case Study:

> The MS Windows family of operating systems allow a specific application to be associated (registered) with any file type. When the user double-clicks on a file with an associated application, the file is automatically opened using that application. Some of the recognized file types are registered at the time the operating system is installed. But none of these are hard-wired into the code. The user has the option to change the application associated with any file type, which allows the functionality of the system to be incrementally expanded. Many applications will also customize the file registration for their own purposes when they are installed. For example, the text processor —it MS Word will claim all files with the extension *.doc* as being in its format. All such files, which in the absence of *Word* would be open in *Windows*' simple text processor *WordPad*, are then automatically opened in *Word*.

□

10.3.2 Logical File Organization

At the highest level, the file system is concerned with only two kinds of files—directories and ordinary (non-directory) files. For directories, it maintains its own internal organization to facilitate efficient search and management of the directories' contents. For ordinary files, the file system is concerned either with the delivery of the files' contents to the calling programs, which can then interpret these according to their type, or with the writing of information to files. Traditionally, a file system views any file as a sequence of **logical records** that can be accessed one at a time using a set of specific operations. These operations are also referred to as **access methods**, since they determine how the logical records of a given file may be accessed.

A logical record in a file is the smallest unit of data that can read, written, or otherwise manipulated by one of the access method operations. Several different file organizations are possible, depending on the answers to the following questions:

1. What constitutes a logical record?
2. How does an operation address individual records?

Logical Records

Logical records may be of **fixed** or **variable length**. In the first case, a logical record can be a single byte, a word, or a structure of arbitrarily nested fixed-length components. But regardless of the complexity of each record, all records within a file must have the same length. In the second case, records may have different lengths; the size or length of a record is typically recorded as part of each record so that access method operations can conveniently locate the end of any given record. Fixed-length record files are easier to implement and are the most common form supported by modern operating systems.

Record Addressing

To access a record within a file, the operation must address it in some way. This can be done either **implicitly** or **explicitly**. For implicit addressing, all records are accessed in the sequence in which they appear in the file. The system maintains an internal pointer to the current record that is incremented whenever a record is accessed. This **sequential** access is applicable to virtually all file types. Its main limitation is that a given record i can only be accessed by reading or scanning over all preceding records 0 through $i - 1$.

Explicit addressing of records allows **direct** or random access to a file. It can be done either by specifying the record's **position** within the file or by using a specific field within the record, called the **key**. In the first case, we assume that the file is a sequence of records such that each record is identified uniquely by an integer from 1 to n (or 0 to $n - 1$), where n is the number of logical records comprising the file. An integer within that range then uniquely identifies a record. In the second case, the user must designate one of the fields within a record as the key. This can be a variable of any type but all key values within a file must be unique. A typical example is

Figure 10.2: Record Types

a social security number. A record is then addressed by specifying its key value.

Figure 10.2 illustrates the four basic organizations graphically. Figure 10.2(a) shows a fixed-length record file. Figure 10.2(b) shows a variable-length record file, where the first field within each record indicates its size. Figures 10.2(c) and (d) show files where each record is identified by a unique key. The first consists of fixed-length records, while the second permits variable-length records, where the record length is again recorded as the first field of each record.

Sequential access to records is equally easy in all four designs. However, only the fixed-length types allow efficient direct access, since the position of any given record can be computed given its index and the record length. For cases (b) and (d) in the figure, additional support structures, such as secondary indices, would have to implemented to make direct access possible.

While variable-size and key-based records are important for many commercial applications, few modern operating systems support them at the file level. The reason is that such complex file organizations have been made obsolete by specialized **database systems** that are better equipped to deal with organizing and presenting data in different ways and from different perspectives. The notable exceptions are file directories. These can be viewed as key-based record files, where the keys are the symbolic file names. The implementation of file directories is discussed in Section 10.4.3.

Memory-Mapped Files

The operations used to read or write files are usually very different in format and functionality from those used to read or write main memory. To eliminate these differences, and thus simplify the task of programming, some systems allow files to be **mapped** directly into the process' virtual memory. Memory-mapping a file involves first reserving a portion of the virtual memory for the file; then, instead of opening the file, the file's contents are equated with the reserved virtual space.

Consider, for example, a file X, consisting of a sequence of n bytes. Assume it is mapped into a process' virtual memory starting at some chosen virtual address *va*. Then, instead of using a special file access operation, such as *read(i, buf, n)*, where i is the open file index and n is the number

of bytes to be moved into the virtual memory area *buf*, the user simply reads or writes the corresponding virtual memory locations directly. That is, accessing locations va through $va + n - 1$ corresponds to accessing bytes 0 through $n - 1$ of the file.

This more convenient view of files is implemented by dynamically linking the file into the process' virtual address space. In systems with segmentation, the file becomes a new segment. In paged systems, the file is assigned one or more entries in the page table, depending on the file size. These pages are handled in the same way as any other pages belonging to the process. That is, each page table entry points to a page frame if the page is currently resident in main memory. It also points to a disk block from which the page is loaded when referenced, and into which its contents are saved when the page is evicted from memory. For read-only code pages, the disk block belongs to the original binary code file. For data and stack pages, the disk block belongs to a special paging or swap file (or a dedicated area on disk). For memory-mapped files, the disk block belongs to the mapped file. Thus accessing a memory-mapped file can be handled with exactly the same underlying mechanisms used for paging.

Case Studies:

1. *Linux.* Linux (and Unix) provides the system call *mmap()*, which allows a process to map a file into its virtual address space. The function takes the following parameters:

   - A file descriptor of the file to be mapped
   - An offset within the file specifying the first byte to be mapped
   - The number of bytes to be mapped
   - A set of flags and access permissions

   The function returns the starting address of the mapped file portion, which may then be accessed using regular virtual addresses. When the process no longer needs the file or wishes to change the size of the mapped area, it issues the call *munmap*. This takes the position of the first byte and the number of bytes to be unmapped, and changes/removes the mapping accordingly.

2. *Windows 2000.* In Section 9.3.1 we presented the idea of section objects, which may be used to share portions of virtual memory among

different processes. The same mechanisms may be used to map files into the virtual address space. When the process creates a new section object, it specifies whether this object is associated with a file. If so, the specified file is located and its contents are mapped into the virtual address space. The file is then accessed as an array. When the process accesses an element in this array, a page fault is generated and the portion of the file corresponding to the page is loaded into memory. Similarly, when the process writes into the array, the data is stored in the file.

□

10.3.3 Other File Attributes

In addition to its name, type, and logical organization, a file is also characterized by a number of different attributes. Some of these are visible and readily accessible at the user level, while others are maintained and used by the file system for its various file and directory management functions. Each operating system implements its own view of what specific attributes should be maintained. The following types of information are generally considered:

Ownership. This records the name of the file's owner. In most systems, the file's creator is its owner. Access privileges are usually controlled by the owner, who may then grant them to others.

File Size. The file system requires current file size information in order to effectively manage its blocks on disk. File size is also important for the user, and hence is readily available at the user level, typically as one of the results of a directory scan operation.

File Usage. For a variety of reasons, including security, recovery from failure, and performance monitoring, it is useful to maintain information about when and how the file has been used. This may include the time of its creation, the time of its last use, the time of its last modification, the number of times the file has been opened, and other statistical information.

File Disposition. A file could be *temporary*, to be destroyed when it is closed, when a certain condition is satisfied, or at the termination of the process for which it was created; or it may be stored indefinitely as a *permanent* file.

Protection. This information includes *who* can access a file and *how* a file can be accessed, that is, the *type of operation*, such as read, write, or execute, that can be performed. Enforcing protection to files is one of the most important services provided by the file system. Chapters 12 and 13 are dedicated exclusively to issues of protection and security.

Location. To access the data in a file, the file system must know which device blocks the file is stored on. This information is of no interest to most users and is generally not available at the abstract user interface. The different schemes for mapping a file onto disk blocks and keeping track of them affect performance crucially and are discussed in Section 10.6.

10.3.4 Operations on Files

The abstract user interface defines a set of operations that the user may invoke—either at the command level or from a program—to manipulate files and their contents. The specific set of operations depends on the operating system. It also depends on the type of files that need to be supported. Below we present an overview of the commonly provided classes of operations.

Create/Delete. Before a file can be used in any way it must be created. The *create* command, which generally accepts a symbolic file name as one of its parameters, creates the file's identity so that it can be referred to in other commands by its symbolic name. The *Delete* or *Destroy* command reverses the effect of the creation by eliminating the file and all its contents from the file system. Since deleting a file by mistake is a common problem with potentially serious consequences, most file systems will ask for confirmation before actually performing the operation. Another common safety measure is to delete any file only tentatively so that it can later be recovered if necessary. For example, the Windows operating system will place a file to be deleted into a special directory called the *Recycle Bin*, where it remains indefinitely. Only when the Recycle Bin is explicitly emptied, for example to free up disk space, is the file deleted irrevocably.

Open/Close. A file must be opened before it can be used for reading or writing. The *Open* command sets up data structures to facilitate the read and write access. An important role of the *Open* command is to set up internal buffers for transferring the file's data between the disk and main memory. The *Close* command reverses the effect of *Open*; it disconnects the file from the current process, releases its buffers, and updates any information maintained about the file.

Read/Write. A *Read* operation transfers file data from disk to main memory, while a *Write* operation transfers data in the opposite direction. The *Read/Write* operations come in two basic forms, *sequential* and *direct*, corresponding to the implicit and explicit forms of record addressing discussed in Section 10.3.2. A direct *Read* or *Write* will access a record that must be designated explicitly by its number (i.e., position within the file), or by its key. A sequential *Read* or *Write* assumes the existence of a logical pointer that is maintained by the system and always points to the record to be accessed next. Each time a sequential *Read* operation is executed, it transfers the current record and advances the pointer to the next record. Thus if the last record read was record i, issuing the same *Read* command repeatedly will access a sequence of consecutive records $i + 1$, $i + 2$, and so on. Similarly, a sequential *Write* command will place a new record at the position of the current pointer. If this position already contains a record, the operation overwrites it; if the pointer is at the end of the file, the write expands the file.

Seek/Rewind. Purely sequential access is too restrictive for many applications. To avoid having to read a file each time from the beginning, a *Seek* command is frequently provided. It moves the current record pointer to an arbitrary position within the file. Thus a *Seek* followed by a *Read* or *Write* essentially emulates the effect of a direct access. How the *Seek* operation is implemented depends on how the file is organized internally, but it can usually be done much more efficiently than reading all records from the beginning. (Note that the *Seek* operation discussed here is a high-level file operation, and should not be confused with a disk seek operation, which moves the disk's read/write head to a specified track.)

A *Rewind* command resets the current record pointer to the beginning of the file. This is equivalent to a *Seek* to the first record of the file.

10.4 File Directories

An operating system generally contains many different files. Similarly, every user often has many files. To help organize these in some systematic way, the file system provides **file directories**. Each directory is itself a file, but its sole purpose is to record information about other files, including possibly other directories.

The information in a directory associates symbolic names given to files by users with the data necessary to locate and use the files. This data

Figure 10.3: Tree-Structured Directory Hierarchy

consists of the various attributes discussed in Section 10.3.3, particularly the information necessary to locate the blocks comprising the file on the disk or other storage media. How this location information is organized and where is it kept is the subject of Section 10.6. In this section we are concerned with the organization of file directories from the user's point of view, and with the operations defined at the user interface to locate and manipulate files.

The simplest possible form of a directory is a flat **list of files**, where all files are at the same level, with no further subdivisions. Such a directory is clearly inadequate for most applications. Thus virtually all file systems support a multi-level **hierarchy**, where a directory may point to lower level subdirectories, which, in turn, may point to yet lower level subdirectories, and so on. Depending on the rules enforced when creating new files and file directories, or when changing the connections between existing ones, we can identify several specific organization within a general hierarchical structure.

10.4.1 Hierarchical Directory Organizations

Tree-Structured Directories

One of the most general practical directory organizations is a **tree** structure. In such a structure, there is exactly one **root** directory, and each file and each directory (except the root directory) has exactly one **parent** directory; the latter is the directory in which the file is listed. The **leaves** of the tree are the data and program files, while all intermediate nodes are directories, sometimes also referred to as **subdirectories**.

Figure 10.3 shows an example of a small tree-structured directory hierarchy. The root directory $D1$ contains two subdirectories, named a and b. Each of these in turn point to lower-level directories (shown as rectangles) or ordinary files (shown as circles.) For example, a in $d1$ points to $D3$ containing c, n, and a; and n points to the file $F2$. The labels Di and Fi are not visible components of the directory structure. They correspond to unique internal identifiers that permit us to refer to directories and files regardless of their position within the directory hierarchy.

Case Studies:

Figure 10.4: Sample Directory Hierarchy in Unix and Windows

1. *Unix.* Figure 10.4(a) shows the top portion of a typical directory structure in Unix. The root directory usually contains a number of directories with agreed-upon names, that point to files where important systems information is kept. For example, */bin* contains executable binary programs corresponding to the various shell commands, such as *cd* (change directory), *ls* (list contents of directory), or *mkdir* (make directory). The directory */dev* refers to files describing specific devices, such as the disk, or the user terminal. The directory */etc* contains programs used primarily for systems administration. */home* is the starting point for all individual user home directories. */tmp* is used for temporary files. Finally, the directory */urs* contains several other subdirectories of executable programs, language libraries, and manuals. Each of the above directories generally contain many files and other subdirectories.

2. *MS Windows.* Figure 10.4(b) shows the top portion of a typical directory structure in the MS Windows operating system. This structure reflects the fact that Windows has been designed for PCs, where each user is in control of the various local hardware devices, rather than for a shared multi-user system. The root of the file system is called the *Desktop* and corresponds to the initial user screen where icons for various subdirectories or programs can be kept for easy access. One of the most important directories on the *Desktop* is *My Computer*, which contains subdirectories for the different storage devices, such as the floppy disk drive (A:), the hard disk drive (C:), and the CD ROM drive (D:). The C: directory, in turn, contains a number of subdirectories, including *Program Files* that store the various applications, and *Windows* which holds the operating system itself. In addition to the storage device directories, *My Computer* generally contains several other directories for accessing and setting up printers, and for various systems management tasks (the *Control Panel*). Other directories appearing on the *Desktop* include the *Network Neighborhood*, which describes other computers on the same network, the *Recycle Bin*, which holds deleted files until they are irrevocably purged from the system, and possibly many other directories and/or programs.

□

Figure 10.5: DAG-Structured Directory Hierarchy

The main attraction of the tree-structured hierarchy is that it is simple to search and maintain. An insertion creates a new entry in the specified directory, which then becomes the root of a new subtree. A delete operation removes the specified entry from the parent directory, together with the corresponding subtree pointed to by the entry. For example, deleting the entry a from the root directory of Figure 10.3 would result in the removal of the entire subtree rooted at directory $D2$.

The main drawback of a strictly tree-structured directory is that *sharing* of files is always *asymmetric*. Any file must appear in exactly one parent directory, which effectively "owns" that file. It is up to the owner to grant or deny access to the file by other users, which must always access it through the owner's directory.

DAG-Structured Directories

Permitting files to have multiple parent directories destroys the tree property but it allows sharing to be done in a symmetrical manner. That is, more than one directory, belonging possibly to different users, can point to the same file and use different symbolic names.

Figure 10.5 show and example of a directory hierarchy structured as a **directed acyclic graph** (DAG) where any file can have multiple parents. Notably, file $F8$ has three parent directories ($D5$, $D6$, and $D3$), two of which refer to it by the same name (p). Directory $D6$ has two parent directories, each at a different level.

A directory hierarchy that allows multiple parent directories for any given file is more complex to maintain than a strictly tree-structured one. First, we need to define the semantics of a delete operation. With only a single parent directory, the file can simply be deleted and the entry removed from that directory. With multiple parent directories, there is a choice. We can remove a file whenever a delete operation is applied to *any* of the parents. This policy is simple to implement, but unfortunately, leaves the references in the other parent directories pointing to a non-existent file, possibly also causing unintended protection violations. The second choice is to remove a file only when it only has a *single parent* directory. For this method, a **reference count** is maintained for each file; it keeps track of the number of parent directories the file appears in at any given time. A delete operation

Figure 10.6: Directory Hierarchy with a Cycle

then removes the file only when its reference count is one; otherwise it only removes the entry from the parent directory.

For example, removing file $F8$ (Figure 10.5) from one or two of the directories, say $D5$ and $D6$, would only remove the corresponding entries, p and e. Only when the file is deleted from the last directory $D3$ would both the entry p and the actual file be removed.

A second problem with allowing multiple parent directories is the possibility of forming **cycles** in the graph. Figure 10.6 shows a structure consisting of the same files and directories as Figure 10.5, but with an additional link from directory $D7$ to its grand parent directory $D2$; this last link completes a cycle in the graph. Cycles cause problems for many of the operations applied to directories. In particular, *searching* for a file may traverse the same portions of the directory structure multiple times, or, may even result in infinite loops. File *deletion* also becomes more difficult because a simple reference count is not sufficient. The reason is that any file that is part of a cycle has a reference count of at least one and thus will not be deleted even if it becomes unreachable from outside of the cycle.

Consider, for example, the deletion of directory $D2$ from the root directory $D1$ in Figure 10.6. The reference count of directory $D2$ is two (it has two parents) and thus the deletion only removes the entry a from directory $D1$, but not $D2$ itself, even though $D2$, $D4$, and $D7$ are no longer reachable from the root or any other directory that's not part of the cycle.

To handle deletion in the case of a cyclic structure requires a **garbage collection** algorithm, which makes two passes over the entire directory structure. During the first pass, all components that have been reached are marked. During the second pass, all unmarked components are deleted.

Most file systems disallow cycles in the directory structure and enforce a DAG hierarchy, in order to eliminate the above problems with deletion and search. This can be accomplished by employing a cycle detection algorithm prior to inserting an existing file or directory in some other existing directory. If this new connection forms a cycle, the insertion is disallowed. Regrettably, such algorithms are expensive to execute, since they may involve traversing large portions of the directory structure.

Figure 10.7: Directory Hierarchy with Symbolic Links

Symbolic Links

To satisfy the need for general file sharing while avoiding the problems with unrestricted directory structures, a compromise solution has been developed. The basic idea is to allow multiple parent directories but to designate one of them as the main (owner) parent. Other directories may refer to the the same file but using only a secondary connection, called a **symbolic link**. Figure 10.7 shows a directory structure with the same files and directories as Figure 10.6, but it allows only one direct parent directory for any file; all other connections must be done by symbolic links, shown as dashed lines in the figure. Thus, the owner structure is a simple tree with real (non-symbolic) links connecting the component parts.

Symbolic links behave differently for different file operations. For reading and writing purposes the file looks the same regardless of whether it is accessed through the main parent directory or one containing only a symbolic link. For example, the name p in directory $D3$ refers to the same file $F8$ in both Figures 10.6 and 10.7. Deletions, however, have a different effect depending on the type of link. For symbolic links, only the link itself is deleted. Only when the delete operation is applied to the actual (non-symbolic) link is the file itself removed. If the deleted file is a directory, the same delete operation is then applied recursively to all files reachable from the deleted directory.

Consider again the deletion of the directory $D2$. Unlike in Figure 10.6, where an unreachable subgraph is created, deleting $D2$ in Figure 10.7 removes not only the entry a from $D1$ but the directory $D2$ itself. It will also remove all other files and directories reachable from $D2$ via actual (non-symbolic) links. In comparison, removing the same directory $D2$ from $D7$, where it is recorded via a symbolic link, will only remove that link, i.e., the entry q.

Case Studies:

1. *MS Windows.* The MS Windows family of operating system implement a form of symbolic links called *shortcuts*. These are pointers to files or folders (directories) that may be arbitrarily copied and moved between different folders and menus. Deleting a shortcut simply discards the pointer.

2. *Unix.* Unix supports a combination of multiple parent directories and symbolic links. Multiple parents are permitted but only with non-directory files. The shared child may then be deleted equally from either parent directory. For all other cross-references, any number of symbolic links may be used. Notably, a directory can only have one parent directory, but it may be referred to by additional symbolic links from any number of other directories.

□

Path Names

All files and directories must be identified uniquely throughout the entire file system. This is usually done by concatenating the symbolic names along the path leading from the root to the desired file, producing a **path name**. The individual components are separated by a special delimiter, such as a period, a slash (/), or a backslash (\). Any path name uniquely identifies one file (or directory) in the file hierarchy. If the hierarchy is a tree, there is only one such path name for any file. With DAG-structured or even more general graph-structured directories, multiple path name may exist for the same file, but, in all cases, a given path name uniquely identifies exactly one file.

For example, file $F9$ in Figure 10.7 can be identified by the two path names */a/t/f* and */z/f*, where, following Unix conventions, the leading slash indicates that the path begins with the root directory and subsequent slashes separate the individual name components along the path.

Note that in the presence of cycles, all files along a cycle can be referred to by an unbounded number of path names. For example, file $F4$ in Figure 10.7 can be identified by the path names */a/l/n*, */a/l/m/q/l/n*, */a/l/m/q/l/m/q/l/n*, and so on; the sub-path *m/q/l*, which repeats one or more times in each path name except the first, is redundant.

While path names uniquely identify any file within the directory structure, their use would be extremely tedious if the user had to specify the full path name each time a file was accessed. To alleviate this problem, the concept of a **current** or **working directory** has been adopted in most file systems. A process can designate, at runtime, any of its accessible directories as its current directory. Files can then be referenced relative to the current working directory. Such path names are referred to as **relative** path names, while those starting with the root of the file system hierarchy are

called **absolute** path names.

For example, if the current directory in Figure 10.7 is $D4$ (identified by the path name *a/l*), then using the relative path name *m/b* is equivalent to using the absolute path name */a/l/m/b*; both refer to the same file $F10$.

To allow a path name to identify a file that is not within the subtree of the current directory, a component of the path name must be able to refer to its parent directory. Let's assume that the special notation ".." is used for this purpose. This notation in conjunction with the path naming conventions allows the user to travel arbitrarily up or down the directory structure. For example, assuming again $D4$ to be the current directory, the two path names *../t/f* and *../../z/f* both refer to the same file $F9$.

10.4.2 Operations on Directories

Similar to operations on files, the user interface specifies a set of operations that the user may invoke—either at the command level or from a program—to manipulate file directories. Some operating systems will permit programs to open, read, write, and otherwise manipulate directories just as ordinary files. But many operations are applicable only to directories. The specific set of operations that are defined depends on the operating system and the structure of the particular file system hierarchy. Below we present an overview of the commonly provided classes of operations.

Create/Delete. A directory must first be created before it can be assigned any contents. Generally, the *Create* command for directories is different from that for ordinary files, since each file type has a different internal format. The difference is even more significant for the *Delete* command. In the case of a directory, the question is what should happen with all the files reachable from this directory. The simplest choice is to allow the deletion of only an *empty* directory, which forces the user to explicitly delete all files listed in the directory to be deleted. The other extreme is to recursively delete all files reachable from the given directory. Many file systems, e.g., Unix, give the user a choice between the different options as part of *Delete* (called *rmdir* in Unix).

List. This operation produces a listing of all files recorded in a directory. For example, listing the directory */b* of Figure 10.7 would show the file names c, n, a and p. Since the purpose of a directory is to keep track of a group of files, the *List* command is one of the most frequently needed operations.

There are typically several parameters or preference settings through which the user can choose how the information should be displayed. In particular, the user may specify the order in which the files should be listed, for example, sorted alphabetically by name, by size, or by some of the dates recorded with the file. The user can also specify how much information about each file should be displayed. In the simplest form, this could be just the lists of the file names, or it could include some or all of the other visible attributes. In most cases, the file type also determines whether the file is included in the directory listing.

Some operating systems support a recursive version of the list command, that displays the contents of the entire subtree rooted at the selected directory. That is, the display shows the contents of the selected directory itself, followed by the contents of all directories reachable from the selected directory.

Case Studies:

1. *Unix.* The Unix *ls* command suppresses by default the display of files whose names begin with a period, as such files are considered system files. But they can be requested by including a parameter (*-a*) with the *ls* command. A number of other parameters are available to control the format of the display and the type of information displayed about each file. A recursive version is also available.

2. *Windows Explorer.* Windows operating systems provide a special application, the *Windows Explorer*, to examine the contents of directories (folders). It consists of a split graphics window, where the left half allows the user to scroll through the directory structure while the right half displays the content of any particular directory selected in the left half. The format and type of information is controlled by specific menus associated with the window.

3. *Windows applications.* Many other Windows applications also display lists of files, for example those that can be opened or saved by that application. But they generally display only those file whose type is associated with the application. For example, the word processor *MS Word* will by default consider files with the extension *.doc*.

□

Change Directory. As explained in Section 10.4.1, most file systems support the concept of a current or working directory to eliminate the need for using absolute path names. The *Change Directory* command permits the user to set the current directory by specifying its absolute or relative path name.

In addition to the current directory, the file system also supports the concept of a **home directory**. This is usually the top-level directory associated with a given user, and is set by default as the current directory when the user logs in. A *Change Directory* command that does not explicitly name any directory also generally defaults to the home directory.

Move. Directory structures must frequently be reorganized. This requires moving files (and directories) between different parent directories. Moving a files between directories changes the file's path name. For example, moving file $F11$ from directory $D7$ to directory $D5$ changes its path name from */a/l/m/c* to */a/r/c*.

Rename. This command simply changes the name under which a file or directory is recorded in its parent directory. Thus, while the user thinks in terms of renaming a file, the operation is really applied to the file's parent directory. For example, renaming file $F1$ from */b/c* to */b/d* changes the entry in directory $D3$. If the *Move* command allows a file to be moved to its new directory under a different name then the *Rename* command can be subsumed by *Move*. The file is simply "moved" to the same directory under a different name.

Change Protection. This command allows the user (generally the owner) of a file to control who can access the file and which type of operation (e.g., read-only, execute-only, read/write) may be performed.

Link. The *Link* operation is applicable to directory structures that allow multiple parent directories for a given file or support symbolic links. In the first case, it creates a new link between a specified file and another existing directory; the new link then becomes the additional parent. In the second case, only a symbolic link (Section 10.4.1) is created.

Assume that the *Link* command for creating symbolic links has the format *SLink(parent, child)*, where *parent* and *child* are arbitrary path names referring to the new parent directory for the given child file, respectively. Then *SLink(/a/l/x, /b)* in Figure 10.7 would make a new entry, x, in the

directory $D4$ (i.e., */a/l*), which would point (via a symbolic link) to the directory $D3$ (i.e, */b*). From then on, the path names */a/l/x* and */b* would refer to the same directory $D3$.

Find. When the directory structure becomes large and complex, finding a specific file can become a challenge. Thus virtually all file systems will support some form of a *Find* command to facilitate this task. The user generally specifies the starting point for the search. This can be the current directory, the working directory, or any other directory. The possible search criteria then include the file name, which could be specified using wild card characters for partial matches, and combinations of other visible attributes, such as the file's size, type, and the various dates recorded with the file.

The *Find* command may be invoked explicitly from the user interface or implicitly as part of various **search routines** invoked by the OS when a file needs to be located. Specifically, when the user names a program to be executed, the file system can follow a set of conventions in attempting to locate and invoke the program.

Case Study:

> Unix allows the user to fully specify the directories to be searched when a program is to be invoked. This is accomplished by defining a series of directories to consider during the search. The path names of these directories are listed in a special system file in the user's home directory containing various user preferences (the file *.cshrc*). These directories are then examined in the given order until a match for the file name is found or the list of directories is exhausted, in which case the search has failed.

□

10.4.3 Implementation of File Directories

Directories must normally be stored on secondary memory because there is potentially a large number of them and each can contain many entries. Most file system treat file directories just like regular files for the purposes of reading and writing. However, their internal organization plays an important role in facilitating efficient access to files. Any directory contains a collection of entries, each corresponding to one file or subdirectory. The two most important questions that need to be answered are:

1. What information should be maintained in each directory entry?

2. How should the individual entries be organized within a directory?

The minimal information that must be recorded within each entry is the file's *symbolic name* and a pointer or *index* to additional descriptive information. At the other extreme, all attributes pertaining to a given file could appear in the directory entry. The disadvantage of the first method is that we need an additional disk operation to access the descriptive information. The disadvantage of the second is that directories could become very large and thus more difficult to manage. Furthermore, the entries could be variable in length. Thus, it would be necessary to support a file structure consisting of complex variable-size records.

But even with the first minimal approach we face the problem of having to manage symbolic file names that can vary in length. If the length is limited to a small number of characters, then each entry can reserve space for the maximum name length. This is the technique used by most older systems, where file names were limited to eight or some other small number of characters. Reserving the maximum space for long name is wasteful. One possible solution is to reserve a relatively small fixed space for each name, followed by a pointer to an overflow heap for long names.

The second question above concerns the internal organization of file entries within a given directory file. First, the directory management routines must be able delete and insert entries efficiently. Second, these entries are accessed associatively by the symbolic file name and thus must support search by content. The simplest approach organizes the entries as an unsorted **array** of fixed-size entries. When variable-size entries are required, an array of variable-size entries or **linked list** can be used instead. Insertions and deletions are simple with both approaches, but searches require sequential scans for a given match. When the number of entries is small, this may be acceptable. But if the list of files is expected to be very long, a more efficient scheme is necessary in order to speed up searches. One possible alternative implements the directory as a **hash table**. Search, insertion, and deletion are then quite efficient. A difficulty here is deciding on an appropriate size for the hash table, a decision that depends on the number of file entries to be managed.

Another alternative is to implement directories as **B-trees** [Comer, 1979]. A B-tree is organized as follows. Every node may be viewed as containing s slots, each capable of holding one data entry and $s + 1$ pointers to nodes

Figure 10.8: B-Tree

at the next lower level of the tree. At any given time, each node must be at least half full, i.e., contain at least $s/2$ entries and the corresponding number of pointers. A data entry consists of a record **key**, such as a file name, and other information, for example, file attributes or pointers to them.

Example:

Figure 10.8(a) shows an example of a B-tree with $s = 2$. The nodes are organized according to the following rule: for any given data entry with key k, the subtree pointed to by the left-hand pointer contains only entries whose key values are *less* than k according to some chosen ordering, such as lexicographical; the subtree pointed to by the right-hand pointer contains only entries with key values *greater* than k. This determines a simple algorithm for locating a record given a key k. Starting with the root node, k is compared to the entries recorded in that node. If a match is found, the corresponding record is accessed. Otherwise, the search follows the pointer immediately succeeding the smallest entry k' found in that node, such that $k' < k$. This is repeated until the desired record is found. The cost of the search operation is proportional to the depth of the tree, i.e., the logarithm of the file size. For example, to locate the record with $k = dg$, first the pointer to the right of entry ck is followed and then the pointer to the left of entry eb is followed. The desired entry dg is found in the leaf node. □

An important property of B-trees is that they are balanced; that is, the distance between the root and any leaf is a constant. The insertion and deletion operations defined on B-trees are guaranteed to preserve this property. Each of these operations may require the splitting or collapsing of nodes on the path between the root and the affected node. However, the cost of insertion or deletion is still proportional to the logarithm of the file size. Figure 10.8(b) shows the result of inserting the entry dh and deleting the entry gf in the tree of Figure 10.8(a).

Unfortunately, the performance of sequential operations on the simple B-trees is not very satisfactory. This is because logically adjacent keys may be stored arbitrarily far apart in the tree structure, and there is no easy way of determining the location of the logically next key, given a key k. In fact, accessing the logically next key is as costly as performing a direct access.

To improve the performance of sequential access, variations of the simple B-tree have been proposed, the most popular of which are B^+*-trees*. The

Figure 10.9: Organization of BFS directories

main distinction is that a B^+-tree keeps all the keys only in the leaf nodes, which are linked together using a simple linked list. The non-leaf nodes form a regular B-tree, whose sole purpose is to serve as index for locating the desired leaf nodes. Search, insertion, and deletion in B^+-trees has the same logarithmic cost as with B-trees. Sequential access, however, is now also efficient: given a key k, the logical successor is found either in the same leaf node (immediately adjacent to the key k), or, if k is the right-most key in that node, by following the appropriate pointer to the neighboring leaf node.

Case Studies:

1. *Unix.* The Berkeley Fast File System (BFS) allows file names of up to 255 characters. Each directory is organized as a variable-size array, where each element describes one file. It records the file name, the file type (directory or regular file), and a number (called the i-node number), which identifies the descriptive information about the file. Since the directory entries are of variable length, they will, over time, be interspersed with empty spaces (holes), which are also of variable length. The file system uses sequential search for both, finding a given file entry using the file name, and finding a hole to accommodate a new file entry.

   To avoid having to maintain a separate list of holes, the file system uses the following convention: every file entry is followed by a single hole (possibly of size zero). The hole size is recorded as part of each entry. This effectively turns each directory entry into a pair, consisting of the file information and an (optional) hole. Figure 10.9 illustrates the idea. It shows the entry for a file named *abc*. When searching for a given *file name*, say *abc*, the system sequentially scans all entries using the length indicators $l1$ and comparing the file name recorded in each entry with the string *abc*. When searching for a *hole* of a given size n, the system scans the entries using again $l1$, but instead of looking for a string match, it looks for an entry where $l1 - l2 \geq n$.

2. *Windows 2000.* The native file system, *NTFS*, of Windows 2000 describes each file and each directory using a descriptor of 1 KB. The contents of a directory may be organized in two different ways, depending

Figure 10.10: Organization of NTFS directories; (a) short directory (b) long directory

on its length. For short directories, the file entries (each consisting of a file name and an index to a file descriptor) are kept inside the directory descriptor itself. Figure 10.10(a) illustrates the situation for a directory $D1$, which contains an entry for a file *abc*. When the number of file entries exceeds the length of the 1 KB descriptor, additional block sequences (called *runs*) are allocated to hold the file entries. These runs are organized in the form of a B^+-tree. Figure 10.10(b) illustrates the situation for a directory $D1$. File entries beginning with the letter a, including the file *abc*, are in the first additional run.

□

10.5 Basic File System

The main functions of the Basic File System are to **open** and **close** files. Opening a file means to set it up for efficient access by subsequent read, write, or other data manipulation commands. Closing a file reverses the effect of the open command and occurs when a file is no longer needed. These and other operations require some basic file data structures or descriptors.

10.5.1 File Descriptors

Section 10.3 introduced the various types of descriptive information associated with a file, including its name, type, internal organization, size, ownership and protection information, and its location on secondary memory. Where this information is kept is an important design decision. One extreme is to maintain the descriptive information dispersed throughout different data structures based on its use by different subsystems or applications. For example, some information can be kept in the file's parent directory, some on a dedicated portion of the disk, and some with the file itself. The other extreme, a cleaner solution, keeps all descriptive information about a file segregated in a separate data structure, pointed to from the parent directory. This data structure is generally referred to as the **file descriptor**.

Case Studies:

1. *Unix.* All versions of Unix file systems provides a comprehensive descriptor for every file and directory in the system. This descriptor is famously, but somewhat obscurely, called an **i-node**. (The "i" stands for "index" as in "file index".) The information kept in each i-node includes the following:

   - identification of the owner
   - file type (directory, regular file, special file)
   - protection information
   - the mapping of logical file records onto disk blocks
   - time of creation, last use, and last modification
   - the number of directories sharing (pointing to) the file

   All i-nodes are maintained in a dedicated table on disk. Directories then contain only the symbolic file names together with the number of the corresponding i-node. The number is used as an index into the table of i-nodes to retrieve the descriptive information. This convention permits the use of large descriptors and supports convenient file sharing by allowing multiple directory entries to point to the same i-node.

2. *Windows 2000.* The NTFS file system keeps all descriptors in a special file, called the **Master File Table (MFT)**, which may be located anywhere on the disk. Each MFT entry is a 1 KB record and describes one file or directory. The first 12 entries are reserved for special files (called *meta* files), such as the root directory, a bit map to keep track of free and occupied disk blocks, a log file, and various other system files necessary to maintain the file system. The remaining entries are available for user files and directories. Each entry is organized as a collection of attribute/value pairs, one of which is the actual content of the file or directory. Other attributes include the file name (up to 255 characters), time stamps, and access information. The location of the actual file's content depends on its length, as has already been explained in the context of directories (Figure 10.10): For short files/directories, the content is kept within the descriptor itself. For longer files/directories, the descriptor contains a list of block numbers (runs) that contain the file/directory content. For extremely large files, i.e., when the number of runs exceeds the length of the descriptor, additional 1 KB descriptors can be allocated and chained to the first descriptor.

□

Opening and closing of files involve two components of the file descriptor: (1) protection information in order to verify the legality of the requested access; and (2) location information necessary to find the file's data blocks on the secondary storage device.

10.5.2 Opening and Closing of Files

Regardless of how the descriptive information about a file is organized, most of it is maintained on disk. When the file is to be used, relevant portions of this information is brought into main memory for efficient continued access to the file's data. For that purpose, the file system maintains an **Open File Table** (OFT) to keep track of currently open files. Each entry in the OFT corresponds to one open file, i.e., a file being used by one or more processes.

The OFT is managed by the **Open** and **Close** functions. The *Open* function is invoked whenever a process first wishes to access a file. The function finds and allocates a free entry in the OFT, fills it with relevant information about the file, and associates with it any resources, such as read/write buffers, necessary to access the file efficiently. Some systems do not require an explicit *Open* command, in which case it is generated implicitly by the system at the time the file is accessed for the first time.

When a file is no longer needed, it is closed either by calling the *Close* command, or implicitly as the result of a process's termination. The *Close* function frees all resources used for accessing the file, saves all modified information to the disk, and releases the OFT entry, thereby rendering the file inactive for the particular process.

The implementation of the *Open* and *Close* functions vary widely with different file systems but the following list gives the typical tasks performed in some form as part of these functions.

Open

- Using the protection information in the file descriptor, verify that the process (user) has the right to access the file and perform the specified operations.
- Find and allocate a free entry in the OFT.
- Allocate read/write buffers in main memory and other resources as necessary for the given type of file access.

Figure 10.11: OFTs in Unix

- Fill in the various components of the OFT entry. This includes initialization information, such as the current position (zero) of a sequentially accessed file. It also includes relevant information copied from the file descriptor, such as the file length and its location on the disk. Additional runtime information, such as the pointers to the allocated buffers or other resources, is also placed into the OFT entry.

- If all of the above operations are successful, return the index of or the pointer to the allocated OFT entry to the calling process for subsequent access to the file.

Close

- Flush any modified main memory buffers by writing their contents to the corresponding disk blocks.

- Release all buffers and other allocated resources.

- Update the file descriptor using the current data in the OFT entry. This could include any changes to the file length, allocation of disk blocks, or usage information (date of last access/modification).

- Free the OFT entry.

Case Study:

The Unix operating system has two levels of OFTs. A shared system-wide OFT is maintained by the kernel, and additional private OFTs are implemented for each user by the I/O libraries. The system-wide OFT supports unbuffered access to the file, while the private OFTs support buffered access. Figure 10.11 illustrates the relationship between the two types of OFTs, the file directories, and the table containing the file descriptors—the i-node list.

Consider first the command for opening a file for low-level, unbuffered access:

```
fd = open(name, rw, ...)
```

The *open* function first searches the file directory for the symbolic name, say F_j in Figure 10.11, and verifies that the requested type of accesses is permitted by reading the corresponding $i\text{-}node_j$. If the latter steps were successful, the function makes an entry in the system-wide OFT and returns the index of that entry, *fd*, to the caller. As already described in Section 10.2, the OFT entry contains the information necessary to locate the file and to read or write sequences of bytes using the commands:

```
stat = read(fd, buf, n)
stat = write(fd, buf, n)
```

These functions access the file's data via the system-wide OFT entry *fd*. They read/write n bytes of data to/from the memory location *buf*, and indicate how many bytes were successfully read/written in the variable *stat*. Both operations are unbuffered; consequently, each invocation results in a disk access (or disk cache access, if the block has already been accessed recently.)

Opening a file for buffered access is accomplished with the command:

```
fp = fopen(name, rwa)
```

where *name* is again the file's symbolic name and *rwa* specifies whether the file is to be used for reading, writing, or appending of data.

The function *fopen* creates a new entry in the process' private OFT and associates a read/write buffer with the entry. It then calls the low-level *open* function, which creates a new entry in the system-wide OFT, or, if the file is already open by another user, it finds the corresponding entry in the system-wide table. In either case, the index *fd* is entered in the private OFT entry. Thus each entry in the private OFT points to the corresponding file via an entry in the system-wide OFT.

fopen returns to the caller a pointer *fp* to the private OFT entry. This pointer is then used to access the file using one of several functions to perform buffered data access. For example, the function

```
c = readc(fp)
```

returns one character of that file in the variable c and advances the current file position to the next character. The operation is buffered in that the characters are copied from a main memory buffer, which is filled initially using a single low-level unbuffered *read* operation. Subsequent calls to *readc* require no disk access until the current file position moves past the end of the buffer. At that time, another unbuffered *read*

operation is invoked to refill the buffer with the next set of characters from the file.

□

10.6 Device Organization Methods

Space on secondary memory devices is organized into sequences of **physical blocks** (or physical records), where a block is typically the smallest amount of data that can be read or written using one I/O operation. The block size is determined by the characteristics of the storage medium. Several logical file records may be mapped onto one physical block, or, conversely, one logical file record may be spread over a number of physical blocks. This ratio is called the **blocking factor**. For the purposes of this section, we assume a blocking factor of one, i.e., each physical block holds exactly one logical file record.

Disks and tapes are the most frequently used secondary storage media. While the blocks on tapes are by necessity arranged and addressed sequentially, disks are two-dimensional surfaces (sometimes with multiple surfaces), where each block is identified by a track number and a sector number within the track. Dealing with physical disk addresses is the task of the I/O system and is treated in Chapter 11. From the file system's point of view, a disk is considered a *one-dimensional sequence of logical blocks* by simply numbering all disk blocks from 0 to $n-1$, where n is the total number of blocks comprising the disk. This abstraction permits the file system to view both disks and tapes as linear sequences of blocks identified by their block numbers. In the remainder of this section we are concerned primarily with disks and thus will use the term **disk blocks** to refer to the sequentially numbered logical blocks provided by the I/O system as the abstract interface to the file system.

The primary task of the Device Organization Methods is then to decide which disk blocks are to be allocated to which logical file records, and to facilitate access to these as required by the read, write, and other data manipulation operations. This information is recorded in the file descriptor introduced in the previous section.

Figure 10.12: File Organization on Disk

10.6.1 Contiguous Organization

A file may be mapped onto a sequence of adjacent disk blocks. In this case, the information needed to locate the disk blocks, which is maintained in the file descriptor, is very simple. It consists of the number of the first block and the total number of blocks comprising the file. The mapping of a single file starting at block 6 and consisting of 4 blocks is shown in Figure 10.12(a).

The main attraction of a contiguous organization is the simplicity with which both sequential and direct access (with fixed-length records) may be accomplished. In particular, sequential access becomes very efficient. That's because adjacent blocks are mapped to the same track or to neighboring tracks of the disk, thus minimizing both seek time, i.e., the need to move the read/write head, and the rotational delay of the disk. Contiguous mapping is particularly attractive for read-only (input) or write-only (output) files, where the entire file is sequentially read or written. In the case of devices that permit only sequential access, such as magnetic tapes, contiguous organization is the only scheme that may be implemented efficiently.

The major problem with contiguous allocation of disk blocks is its inflexibility in deleting and inserting records. For variable-length records, changing the length of a record is also problematic. In all these cases, it may be necessary to physically move all records following or preceding the location of the modification in order to preserve the contiguity of allocation. Alternatively, insertions and deletions may be permitted only at the end of a file.

A related problem is how to cope with files that may expand in size due to insertions or appends of new records. One solution is to declare the maximum file length *a priori*. Unfortunately, if too little space is allocated, the file may not be able to expand, or it may have to be moved to another area of the disk where enough contiguous blocks are found. Too much space, on the other hand, results in wasted resources.

Finally, a contiguous allocation fragments the disk into sequences of free and occupied blocks. This is similar to the problem of managing variable-size partitions in main memory (Section 7.2.2). The file system must keep track of the available holes and allocate them to files using schemes similar to those for main memory management, such as first fit or best fit.

10.6.2 Linked Organization

With this organization, the logical file records may be scattered throughout the secondary storage device. Each block is linked to the logically next one by a forward pointer, as illustrated in Figure 10.12(b), which again shows a file starting at block 6 and comprising a total of 4 blocks. pointer.

The main advantage of a linked organization is the ease with which records may be inserted or deleted, thus expanding or contracting the file without excessive copying of the data. There is also no need to impose an upper limit on the file size. With a linked organization, sequential access is also easy but much less efficient than with a contiguous organization. The reason is that disk blocks must be read one at a time, since the number of any block is known only when its predecessor has been read. Thus each block will experience the average rotational delay of the disk. Furthermore, each block may potentially reside on a different track and thus require a disk seek operation.

Another problem is that direct access is very inefficient. To access a given block requires that the read or write operation follows the chain of pointers from the start until the desired block is found. Yet another serious concern with the linked organization is reliability. If a disk block becomes damaged and cannot be read, the chain of pointers is broken; it is then difficult to find the remainder of the file.

Variations of the Linked List Organization

Many of the drawbacks of the simple linked organization can be alleviated by keeping the block pointers segregated in a separate area of the disk, rather than maintaining them as part of the disk blocks. This can be accomplished by setting up an array, say $PTRS[n]$, where n is the total number of blocks comprising the disk. Each entry $PTRS[i]$ corresponds to disk block i. If block j follows block i in the file then the entry $PTRS[i]$ contains the value j of the corresponding entry $PTRS[j]$.

The pointer array is maintained on a dedicated portion of the disk, for example, the first k blocks of track zero. The number k depends on the total number of disk blocks n, and the number of bits used to represent each entry $PTRS[i]$. For example, assume that a 4-byte integer is needed to record a block number. If the block size is 1024 bytes, $1024/4 = 256$ block numbers can be accommodated in each of the pointer blocks. With $k = 100$, the number of data blocks the $PTRS$ array can keep track of is 256,000. Thus

n would be $256,000 + k = 256,100$.

This organization has been adopted by a number of PC-based operating systems, including MS-DOS and OS-2. Figure 10.12(c) illustrates the idea. It shows the same file as Figure 10.12(b) but with the pointers segregated within the first three blocks of the disk. Thus $PTRS[6] = 18$, $PTRS[18] = 11$, $PTRS[11] = 13$, $PTRS[13] = NULL$. One advantage of this scheme over the simple linked list organization is that sequential access can be performed much more efficiently. All pointers that need to be followed are concentrated within a small number of consecutive blocks containing the $PTRS[n]$ array, and can thus be read with a single read operation. Furthermore, these blocks can usually be cached in main memory, further improving the time to locate the blocks of a given file.

A serious limitation of the linked list organization (Figures 10.12(b) or (c)) is that the individual blocks are scattered throughout the entire disk. This unnecessarily increases the number and duration of disk seek operations. An *optimized* version of a linked list organization tries to keep all blocks clustered close together, preferably on a single cylinder or a small number of adjacent cylinders. An additional optimization is achieved if sequences of adjacent blocks are linked together, rather than individual blocks. Figure 10.12(d) illustrates this idea. It shows the file allocated in two block groups linked together, one consisting of blocks 6 through 8, and the other consisting of the single block 11. This organization combines the advantages of the contiguous and the individually linked organizations.

10.6.3 Indexed Organization

The purpose of indexing is to permit direct access to file records while eliminating the problems of insertion and deletion inherent to schemes with contiguous allocation of storage. The records may be scattered throughout the secondary memory device just as with the linked organization. An **index table** keeps track of the disk blocks that make up a given file.

There are several forms of indexing depending on how the index tables are organized and where they are kept. In the simplest case, each index table is just a sequence of the blocks numbers. Figure 10.12(e) is an example where the index is implemented as part of the file descriptor. It shows a file consisting of the same initial blocks as the previous linked organizations.

Since the size of the descriptor is significant, some file systems store the index table as a separate block. The table can reside anywhere on the disk just like any other block. The descriptor itself then only contains the number

Figure 10.13: File Mapping in Unix

of the index block.

Variations of Indexing

The principal limitation of the simple index scheme is the fixed size of each index table. This size determines the maximum number of blocks a file can occupy. The problem can be alleviated by keeping track of **groups** of adjacent blocks, rather than individual blocks, as was done in the case of the linked organization of Figure 10.12(d). Nevertheless, the number of possible entries remains fixed. Thus the table must be large enough to accommodate the largest possible file. Consequently, much space is wasted on unused entries for smaller files. Several solutions exist for the problem. One is to organize the indices as a **multi-level hierarchy**. For example, in a 2-level hierarchy, each of the entries of the root, or **primary index**, would point to a leaf, or **secondary index**. Entries in the secondary indices then point to the actual file blocks. This makes the number of secondary indices, and hence the space needed to manage a file, variable, depending on the file size.

The disadvantage of the multi-level hierarchy is that the number of disk operations necessary to access a block increases with the depth of the hierarchy. An **incremental indexing** scheme minimizes this overhead while allowing the index to expand based on the file size. This scheme allocates a fixed number of entries at the top index level for actual file blocks. If this is not sufficient to accommodate a file, the index can be expanded by additional, successively larger indices. The first expansion is just another list of n additional blocks. The second expansion is a 2-level tree, providing for n^2 additional blocks. The third expansion is a 3-level tree, indexing n^3 additional blocks, and so on.

The main advantage of the technique is that the overhead in accessing a file is proportional to the file size. Small files, which constitute the majority of existing files, are accessed efficiently without indirection. At the same time, it is possible to construct extremely large files at the cost of additional indirect accesses.

Case Studies:

1. *Unix.* A 3-level incremental indexing scheme has been implemented in the Unix operating system. Each file descriptor, or i-node in Unix

terminology, contains thirteen entries describing the mapping of the logical file onto disk blocks (Figure 10.13). The first ten entries point directly to blocks holding the contents of the file; each block comprises 512 bytes. If more than ten blocks are needed, the 11th entry of the index table is used to point to an indirection block which contains up to 128 pointers to additional blocks of storage. This yields a total of (10+128) blocks to hold the file's content. If this is still insufficient, the 12th entry of the index table points via a twofold indirection to additional 128^2 blocks of storage. Finally, for files exceeding the total of $(10 + 128 + 128^2)$ blocks, a threefold indirection starting in the 13th entry of the index table provides 128^3 blocks of additional storage to hold the file.

2. *Windows 2000.* The NTFS file system discussed in Section 10.4.3 implements a 2-level hierarchy. The contents of short files (or directories) are kept directly in the file descriptor. For longer files, the descriptor contains lists of block numbers where the actual contents are kept.

□

10.6.4 Management of Free Storage Space

When a file is created or when it expands, the file system may have to allocate new disk blocks to it. Deciding which blocks to allocate is important since the decision determines future disk access times. Similarly, when a file is deleted or shrinks in size, some blocks may be released and added to the pool of free disk space.

Allocating disk blocks and keeping track of the free space is one of the main functions of the Device Organization Methods of the file system. The disk space is generally shared by many different files. Not surprisingly, some of the techniques for main memory administration described in Chapter 7 are also applicable here. There are two principal approaches to keeping track of available disk space.

Linked List Organization

We can treat all free blocks as just another file. This file can then be organized using the same techniques presented earlier in this section for keeping track of file blocks. In particular, the simple linked-list method (Figure 10.12(b) or (c)) could be applied, where each free block contains

a pointer to its successor on the linked list. The main drawback is that it does not facilitate block clustering. Furthermore, efficiency in allocating and releasing of blocks is also poor. It is frequently the case that multiple blocks need to be allocated or released at the same time. The simple linked organization requires a separate disk access operation for each block. Similarly, adding several blocks to the free list requires that each is written separately.

A much better method is to link together **groups** of adjacent blocks instead of individual blocks. This is analogous to the modified linked list approach of Figure 10.12(d). The segregation of adjacent blocks not only shortens the linked list, but makes it much easier to allocate (and free) multiple blocks using a single disk access operation.

Bit Map Organization

Since disk space is allocated naturally in fixed-size blocks, the status of the disk space can be conveniently maintained as a bit map where each bit represents the state (free or occupied). Because only one additional bit is required for each disk block, very little disk space is wasted. For example, if disk storage is allocated in blocks of 512 bytes, the bit map overhead is only $100/(512 \times 8) = 0.024\%$.

Also, sequences of adjacent free blocks and occupied blocks are naturally segregated in the bit map as groups of zeros and ones. Consequently, all operations, including finding free blocks in a given vicinity, allocating blocks, and releasing blocks, can be implemented efficiently using bit-manipulation operations, as developed in the context of main memory management (Section 7.2.2.)

10.7 Principles of Distributed File Systems

In the the previous sections, we assumed a single centralized file system. This system manages all files and directories on the underlying storage devices—typically a disk or a set of disks, and provides the user interface.

Networking makes it possible to connect many different computers together, each of which can have its own secondary storage subsystem. A typical environment is a local area network, consisting of some number of PC's and diskless workstations, as well as workstations with local disks. One function and goal of the file system in such a distributed environment is to present a unifying view of all files to the user and, as much as possible, to hide the fact that they reside on different machines. Another key function

of a distributed file system is to support the sharing of files and directories in a controlled manner.

Most operating systems will provide some utilities to share files *explicitly*, that is, to copy them between different computers, each of which maintains its own (centralized) file system. One of the most common utilities to copy files between machines is *ftp* (file transfer program), which uses the Internet's standard File Transfer Protocol (FTP) to accomplish its task. *ftp* opens a dialog session between its invoking client and a server on a remote machine. It allows the user to execute a number of useful file and directory manipulation commands, including the transfer of files between the client and server machines.

Another common utility, supported on Unix and Linux machines, is *rcp* (remote file copy). Similar to the local file copy command (*cp*), *rcp* requires two parameters, the source and the destination of the file to be copied. The main difference between *cp* and *rcp* is that the source or destination parameter of the latter may be a local file or a remote file, specified as *hostname:path*, where *hostname* is the name of a remote machine and *path* is the file's path name on that machine.

Such copying tools, while extremely useful, do not hide the distribution of files and thus require that users know about the files' locations and names. A **distributed file system** (DFS) makes this distribution transparent by presenting a single view of all files and providing easy access to them regardless of their present location.

10.7.1 Directory Structures and Sharing

Similar to centralized systems, a distributed file system must define a directory structure, together with a set of directory operations, to permit users to locate files. It also provides a high-level, abstract view of files, represented by a set of operations to read, write, and otherwise manipulate the files' contents. However, a DFS must extend these structures, views, and services because files and directories can reside on more than one machine.

There are many ways to define a distributed file system, differentiated by how they handle the following important issues:

1. **Global vs. Local Naming.** Is there a single global directory structure seen by all users, or can each user have a different local view of the directory structure?

Figure 10.14: Unifying file systems (a) two separate file systems (b) creating a common root (c) mounting (d) shared subdisrectory

2. **Location Transparency.** Does the path name of a file reveal anything about the machine or server on which the file resides or is the file's location completely transparent?

3. **Location Independence.** When a file is moved between machines, does its path name need to change or can it remain the same?

Let us examine these crucial points in more detail and consider the tradeoffs between the possible options.

Global Directory Structure. From the user's point of view, the directory structure of a DFS should be indistinguishable from a directory structure on a centralized system. In such an ideal directory, all path names would be globally unique; that is, a file would be referred to by the same name regardless of which machine the user was currently residing on, and would not change if the file was moved physically between different servers.

Unfortunately, such an ideal distributed file system is very uncommon. Less convenient and ideal systems are the norm for two important reasons. First, a DFS is rarely developed and populated with files from scratch. Rather, a set of pre-existing file systems is combined into a common federation. The second reason is that there are usually certain files that must reside on specific machines. For example, workstation initialization programs or files that maintain some portion of the machine's description or state may need to reside on that specific machine. Consequently, a distributed file system must leave certain aspects of distribution visible to the user.

The problems can be solved by sacrificing location transparency. In the simplest case, the path name of a remote file is simply prefixed by the name of the machine or server on which the file resides. A common notation for this approach is *m:path*, where *path* is a path name on the specific machine, m. A more elegant solution is to assign a unique name to the root of each local file system and unify them under a common global root.

To illustrate this approach, consider Figure 10.14(a), which shows two file systems, each with its own root (/), and each residing on a different file server, $S1$ and $S2$. Figure 10.14(b) shows a federated directory structure

resulting from combining the two local file systems under the names of their respective servers.

The new directory structure is global, resulting in path names that are equally valid on any machine. For example, the path name */S1/usr/u1* would identify the same file *u1*, regardless of whether it was used on server $S1$ or $S2$. The main problem with using such path names where the slash (/) refers to the new global directory is that path names that were used on the individual systems before they became federated would cease to work properly, since the slash was intended to refer to the *local* root directory. For example, */usr/u1* identifies the file *u1* on the local file system of server $S1$ but is invalid in the global structure (Figure 10.14(b)).

To solve this naming dilemma, the following scheme was developed and implemented in the *Unix United* distributed operating system. The slash is retained as a reference to the *local* root directory, so that existing programs do not have to be modified. To access files above the local root, the same notation (..) is used as for referring to any other parent directory. Thus a path name reveals clearly the boundary of the system but it permits the user to reach any file within the global structure. For example, to reach the file *u1* from *u2*, either of the following path names could be used: *../../../S1/usr/u1* or */../S1/usr/u1*. The first is a relative path name, starting in the current file *u2*, while the second is absolute, starting with the local root (/) of *u2* and moving up to the global root prior to descending down toward *u1*.

Note that combining the original file systems by simply merging their root directories, i.e., without introducing the new root directory, would be difficult due to name conflicts. For example, both file systems have the directory *usr*, each of which may contain files with the same names and thus could not simply be merged into one. The disadvantage of introducing the new directory names *S1* and *S2* is that file names are *not* location transparent. Choosing symbolic names for *S1* and *S2* can hide the identity of the underlying machines, but their position within the hierarchy, combined with the fact that each contains well-known system files such as *usr*, reveals information about the underlying physical file distribution.

Local Directory Structures. Another way to combine multiple existing files systems into one is by a **mounting** mechanism. Thereby, a user (with the necessary systems privileges) can temporarily attach any part of a file hierarchy residing on some remote machine to a directory within the local file system. The chosen local directory is called the **mount point**.

After the mount operation is complete, the remote subdirectory becomes a subtree of the local directory, with the mount point as its root directory. Figure 10.14(c) illustrates this for the two structures of Figure 10.14(a). The directory *mp* is the mount point, over which a user on server *S1* mounts the subtree identified by the name */usr*, residing on the remote server $S2$. Thereafter, all users on $S1$ may use the path name */mp* to refer to directory identified by */usr* on $S2$, and the path name */mp/...* to directly access any files under that directory as if they were local. For example, */mp/u2/x* on $S1$ refers to the same file as */usr/u2/x* on $S2$. Note that the previous contents of *mp*, if any, are inaccessible while the remote subtree is mounted. Thus mounting is typically done over special mount points—empty directories set up for just that purpose.

This approach provides complete location transparency since the path name does not reveal the actual location of any file. The main disadvantage is that different users may have different views of the file system hierarchy. As a consequence, path names are context sensitive; their meaning depends on which machine they are interpreted. For example, a program using the path name */mp/...* on machine *S1* would fail or possibly use a wrong file when moved and executed on machine *S2*.

Shared Directory Substructure. Because certain files often must reside on specific machines, the principles of location transparency and location independence can never be fully realized. An attractive compromise solution to this problem, pioneered by the *Andrew file system* [Howard et.al, 1988; Satyanarayanan, 1990], and adopted by successor, the Coda file system [Satyanarayanan et al, 1990], specifies a two-part hierarchy for each machine. One part consists of local files, which are different for each machine. The other is a subtree of shared files; these are, of course, identical on all machines. Figure 10.14(d) shows the idea for the two file systems of Figure 10.14(a). Each now has a new subdirectory, named *shrd* in this example, under which all files shared by the two systems are maintained. All other files are local. Thus modifying the directory *usr* on either machine would affect only files on that machine, while modifying any portion of the *shrd* subtree would become visible by users on all participating machines.

This solution is a compromise in that it provides a single view of all shared files while permitting each subsystem to maintain its local files. It is also beneficial for performance since many files, such as temporary files, will never be shared and hence do not have to be subjected to the overhead

resulting from file sharing.

10.7.2 Semantics of File Sharing

Files in a distributed environment are no different from files in a centralized system. They are characterized by their name, type, internal organization, and a number of other attributes, as discussed in Section 10.3.3. Distributed files are accessed and manipulated using the same operations as those used in a centralized environment. That is, a file can be created and deleted, and opened and closed; and its contents can be accessed with read, write, and seek operations.

The main difference between centralized and distributed files is their semantics of sharing. Ideally, we would like files to behave the same way in both environments. Unfortunately, performance considerations make it necessary to sometimes relax this equivalence. This leads to the following possible types of file sharing semantics:

- **Unix semantics**. In most centralized file systems, including Unix, sharing a file implies that all write operations become immediately visible to all users sharing that file. That is, a read operation will always see the data written by the last write operation. Implementing the same semantics in a distributed environment requires that all write operations are immediately propagated through the network to the corresponding file, resulting in much network traffic. Furthermore, since network delays are generally unpredictable, it is not always possible to guarantee that write operations are performed in the order in which they were issued on different machines. For these reasons, other forms of file sharing semantics have been developed for distributed file systems.

- **Session semantics**. To avoid the need to propagate all write operations to remote files immediately, a file opened under session semantics is treated as a private copy by each user for the duration of the session. Only when the file is closed are the changes propagated to the original file and thus become visible to other users. This relaxed form of semantics has less overhead than the Unix semantics, but is less appealing due to its increased unpredictability. Notably, when two processes simultaneously update a file, the file's content will depend on which process closes the file last. The changes made by the first

Figure 10.15: Client/Server Architecture of a DFS

closing process are simply lost; that is, they are overwritten by the values written by the last process.

- **Transaction semantics**. Under the session semantics discussed above, all modifications to a file performed by a user remain invisible until the file is closed. To reduce the time during which the changes are invisible, transaction semantics permits the user to designate specific portions of the code to be transactions, that is, segments of code that can be considered indivisible. All modifications of a file performed during a transaction become immediately visible to all other processes sharing the file at the end of the transaction. In other words, a transaction can be used as an atomic operation, whose effects become immediately visible when it terminates.

- **Immutable-files semantics**. The problem of delayed file updates can be avoided entirely by simply disallowing any modifications of already existing files. That is, all files are treated as read-only. A write operation results in the creation of a new file, typically referred to as a new **version** of the file. The problem of file sharing then becomes a problem of version management, for which a number of solutions exist to maintain data consistency.

10.8 Implementing DFS

The Basic Architecture

Most distributed file systems are organized as collections of servers and clients. Servers run on machines that contain portions of the file system, while clients run on machines that access the files and directories in response to users' commands. The subdivision of machines into servers and clients, however, is not absolute; most machines can operate in either capacity, depending on the operations being performed.

Figure 10.15 shows a generic structure of a client/server-based distributed file system architecture, patterned after the popular Sun Microsystems **Network File System**, commonly known as **NFS** [Callaghan, 2000]. The figure shows two machines, A and B, each of which contains a local disk and a portion of the file systems. Let's assume that machine A requests access to a

given file. The request is presented to the top layer of the file system of A, called the virtual file system in the figure. This module determines whether the file is held locally or on a remote server. In the former case, it forwards the request to the local file system, which accesses the file on the local disk. If the file is remote, machine A will play the role of a client. It will determine the appropriate server and send the request through the network to the corresponding machine. The server accesses the file via its local file system and returns the result back to the client.

Most existing distributed file systems are based on the basic client-server architecture. What differentiates the various types is how much functionality is implemented in the client and how much is implemented in the server. The most important issues are the forms of caching employed, the management of the state of open files, and the level of file data replication.

Caching

When a client accesses a remote file it experiences two forms of overhead: a **network delay** and a **disk access delay** on the server. Both can greatly be diminished by caching. The main questions are whether the cache should be maintained on the server or on the client, and what information should be cached.

Server caching. The disk access overhead can be eliminated on subsequent accesses to the same file if the server caches the file in its main memory. One approach caches the entire file. This can be wasteful if only a small portion of the file is being accessed; it also complicates the cache management since each file has a different length. A second option is to cache only the disk blocks that have been accessed. This technique results in a more structured and efficient cache, but requires a block replacement scheme, such as the LRU policy discussed in the context of paging (Section 8.3.1).

The main advantage of caching of file data in the server is that it is easy to implement and can be made totally transparent to the client. Both file-level and block-level caching reduce the need to access the disk and thus improve file access. The principal drawback is that the data still needs to be transferred through the network at each request.

Client caching. To reduce the network overhead, caching can also be performed on the client. Similar to server caching, the client can either download the entire file into its own local memory or local disk, if one is available, or it can cache only those disk blocks it is accessing. Unfortunately, client caching is a source of potential inconsistencies. Whenever two clients share

a file, updates performed to the locally-cached data are not immediately visible to the server or to other process using the same file concurrently. We can subdivide this problem into two related issues: (1) when to update the original file on the server; and (2) when and how to inform other processes sharing the file of the changes that have been made. We examine these two issues in turn:

Write-through versus delayed writing. The simplest possible but least efficient solution to the problem of updating the original file on the server is to propagate each modification to the server immediately. This permits us to enforce Unix semantics, but it greatly diminishes the benefits of client caching. The alternative is to adopt one of the other weaker forms of file sharing semantics as discussed in the last section. For example, session semantics permits an updates to be kept local and propagated to the file server only when the file is closed. Similarly, transaction semantics reduces the frequency with which updates need to be propagated to the server. Thus the basic trade-off is improved performance at the cost of less user-friendly file sharing semantics.

Cache update schemes. Updating the file on the server is only the first part of the file coherence problem. Once this is accomplished, other processes that have cached any portions of the file in their local space must be notified of the changes. This can be initiated by the server or by the clients. In the first case, the server keeps track of all processes that have opened the file for access, and it informs them automatically of any changes to the file. Unfortunately, the scheme violates the basic client-server model—the server, which should only respond to client requests, must in this case take the initiative to contact the client.

If the client is made responsible for validating its own cache, it must decide when to check with the server about any possible changes to the file. Doing so prior to every access would defeat the purpose of caching. On the other hand, checking less frequently increases the danger of reading out-of-date information. Relaxing the file sharing semantics again alleviates the problem. For example, with session semantics, the client only needs to check with the server at the time of opening the file.

Stateless Versus Stateful Servers

Before a file can be used for reading or writing, it must be opened. This causes a new entry in the open file table (OFT) to be created, that stores the current state of the file. The state typically consists of the current

Figure 10.16: Interactions with: (a) a stateful server; (b) a stateless server

position within the file, the access mode in which the file has been open, and location information to find the file's contents on the disk. In the case of a centralized system, the OFT is maintained by the local file system. In a distributed system, the open file table can be maintained in the server or in the client. This leads to two possible forms of a server, referred to as stateful and stateless, respectively.

Stateful servers. A *stateful* server maintains the open file tables for all its clients. When a client wishes to access a file, it sends an open command to the server. The server makes a new entry in the open file table and returns the index of this entry to the client. The index is used by the client for subsequent read or write operations to the file. This procedure is analogous to that for opening a file in a centralized file system.

Figure 10.16(a) outlines the basic interactions between a client and a stateful server. Assume that a user process wishes to open a file X and perform a series sequential read operation on it. It first issues the command i = *open(X, ...)*. The client portion of the DFS passes the command to the corresponding server portion on the host holding the file X. The server opens the file by making a new entry in its OFT. Part of this entry is the current position within the file, *pos*, which is initialized to zero. The server than returns the OFT index i to the user process via the client. Each of the subsequent *read* commands issued by the user is also passed to the server by the client. For each command, the server performs the operation, updates the OFT (notably the current file position), and returns the data (n bytes) to the client; the data is either directly stored or copied into the specified user buffer (*buf*).

The client's role is straightforward. It acts essentially as a conduit between the user process and the server by exchanging the commands and the results between them. The commands used to open and access the file are the same as in a centralized file system. The problem with the approach is reliability. Since the client and the server are on different machines, each can crash independently. All state information about open files is generally held in the server's main memory. If the server crashes, and subsequently recovers, this information is lost, including the current position within the file and any partially updated attributes, such as the file length or disk location. It is then up to the client to reopen the file and to restore the state

of the file prior to the crash. When the client crashes while having open files, the server is left with information in its open file tables that will not be claimed by the client after recovery, and must be discarded through a recovery process.

Stateless servers. A *stateless* server has no state information about any open files. Instead, each client is responsible for keeping its own open file table. Since the server does not accept any open or close commands, each access to the file by the client must contain all the information necessary to perform the operation. In particular, each command must contain the file's symbolic name, which must be maintained as part of the OFT. Each read and write command must also specify the starting position within the file, in addition to the number of bytes to be transferred.

Figure 10.16(b) sketches the interactions between the client and the stateless server using the same sequence of user operations as in the stateful example. The *open* command causes the creation of a new OFT entry. This, however, is maintained by the client; the server is not aware of this operation. The OFT index (i) is returned to the user as before. The subsequent *read* operations cannot simply be forwarded to the server but must be transformed by the client into analogous *read* operations that do not rely on the existence of any state information in the server. In particular, each *read* must specify not only the number of bytes (n) but also the file's symbolic name (X) and current starting position (pos), both of which are recorded in the OFT and maintained by the client.

Because the server has no current state information about any open files, it is possible to recover transparently after a crash—the server simply restarts and continues accepting commands from its clients. The commands are **idempotent**, meaning that they can be repeated any number of times and they always return the same data. The price to pay for this improved fault tolerance is the increased burden placed on the client; more information must be sent with every command than was the case with a stateful server.

Case Study:

> Sun Microsystems Network File System, NFS, implements a stateless server that clients use through a set of remote procedure calls. Most operations of the server, including reading and writing files and directories, are idempotent. Those that are not idempotent cause no inconsistencies in the stored data, but may generate error message that can be confusing to clients. For example, deleting or renaming a file are

not idempotent. If the server's reply to such a command is lost, the client generally retransmits the same command. Since the operation cannot be repeated, the server responds with an error message, such as "file does not exist," even though the operation has been performed successfully.

Caching is performed by both the server and the clients. However, NSF also guarantees Unix semantics, which interferes with client caching. To solve this dilemma, NFS allows client caching only for non-shared files, and disables it when a file is shared by multiple clients. This, unfortunately, introduces some amount of state into the "stateless" server, because the server needs to keep track of which client has cached which file. The popularity of NSF indicates that a successful distributed file system needs to make many important trade-off decisions in its design to satisfy a wide range of requirements.

□

File Replication

Keeping multiple copies of file on different servers has several important benefits:

- **Availability**. When a server holding the only copy of a file crashes, the file becomes inaccessible. With additional replicas, users may continue accessing the file unimpeded.
- **Reliability**. When a file server crashes, some files may be left in an inconsistent state. Having multiple independent copies of a file is an important tool for the recovery process.
- **Performance**. Maintaining multiple copies of files at different locations increases the chances for a client to access a file locally, or at least on a server that is in its close proximity. This decreases network latency and thus improves the time to access the file. It also reduces the overall network traffic.
- **Scalability**. A single server can easily be overwhelmed by a number of concurrent accesses to a file. Maintaining multiple copies of the file defuses the bottleneck and permits the system to easily scale up to a larger numbers of users.

File replication is similar to file-level caching in the client. In both cases, multiple copies of a file are maintained on different machines. The main difference is that caching only maintains a *temporary* copy in the client while the file is being used. With file replication, the copy is *permanent*. Furthermore, a cached copy is often kept on the client's machine, while a replicated file is kept on a separate server that may be a different machine from the client's.

The problem with file replication is again consistency. Similar to caching, the system must guarantee that all update operations are applied to all replicas at the same time or the file semantics must be relaxed. There are two main protocols the file system can follow when reading and writing file replicas to keep them consistent:

Read-any/write-all protocol. As the name suggests, this protocol allows a process to read any of the available replicas. An update, however, must be propagated to all replicas simultaneously. This latter requirement is difficult to implement. First, the update may have to be propagated atomically to all replicas so that it is not interleaved with other update requests, thus possibly introducing inconsistencies. Special group communication primitives can be used for that purpose. When the application domain allows interleaving, the propagation of the updates does not have to be atomic, but it must still be sent to all copies of the file. This raises the question of what should be done when one or more of the copies are currently unavailable, for example, due to a temporary server crash. Forcing the writing process to wait until all copies are available is not an acceptable solution. The alternative is to put the burden on the failed server. When it recovers, it must communicate with other servers and bring all its files up to date.

Quorum-based read/write protocol. The previous read-any/write-all protocol is only a special case of a more general class of protocols called quorum-based protocols. These avoid the need to always update all replicas of a file, but at the expense of having to read more than one copy each time. The number of replicas, w, that a process must write each time is referred to as the **write quorum**. The number of replicas, r, that a process must read each time is referred to as the **read quorum**. The two numbers r and w are chosen to obey the following constraints:

$$r + w > N$$
$$w > N/2$$

where N is the total number of replicas. The first rule guarantees that any read quorum intersects any write quorum. That is, any read operation

Figure 10.17: Accessing files using read/write quorum

will see at least one replica that has been updated by the latest write operation and thus represents the latest version of the file. The second rule prevents two processes from updating two disjoint subsets of the replicas, both of which could end up having the same version number or time stamp. Subsequent read operations then could not differentiate between them.

Example:

Consider a system of seven replicas of a given file, numbered $F1$ through $F7$. If we assume $r = 4$ and $w = 4$, then any read operation and any write operation must access at least four different replicas. Figure 10.17(a) shows an example where a write operation modified replicas $F1, F2, F4$, and $F5$, and the subsequent read operation read the replicas $F2, F3, F5$, and $F6$. The replica $F2$ is in both sets, thereby ensuring that the read operation will access the latest version of the file. Figure 10.17(b) illustrates the extreme case, where the size of the write quorum is 7, while the size of the read quorum is 1. In other words, a write operation must update *all* replicas, while a read operation may read *any* replica. This extreme example corresponds to the previous protocol, called read-any/write-all. □

Contents